Ricorsione, Iterazione e Asserzioni Logiche

26 marzo 2010

1 Ricorsione e Iterazione in `Java`

In questa sezione verranno riviste nozioni relative ai meccanismi base di controllo: iterazione e ricorsione. L'obiettivo è quello di confrontare questi due meccanismi di controllo e introdurre delle metodologie per valutare la correttezza dei programmi.

1.1 La funzione fattoriale

La funzione fattoriale, indicata in matematica con il simbolo !, può essere informalmente definita come segue: il fattoriale di un intero n è il prodotto di tutti i numeri da 1 ad n, cioè $1 \times 2 \times \ldots \times (n-1) \times n$. Tale scrittura non è precisa, perché utilizza i puntini (un simbolo che non appartiene al linguaggio matematico). Al contrario, tale scrittura può essere formalizzata usando la notazione di *produttoria*:

$$n! = \prod_{i=1}^{n} i$$

Similmente alla notazione di sommatoria ($\sum$), la produttoria è usata per indicare il prodotto di un certo numero di numeri; si assume che la produttoria vuota vale 1 (l'elemento neutro del prodotto), così come si assume che la sommatoria vuota è 0 (l'elemento neutro della somma). A questo punto, scrivere un metodo `Java` iterativo che calcola il fattoriale è molto facile e segue lo schema di programmazione necessario per scrivere un programma che calcola una sommatoria. Si eseguono i prodotti accumulando i risultati parziali in una variabile. La variabile viene inizializzata ad 1, l'elemento neutro del prodotto.

```
public static int fattIt(int n) {
   int f=1; /* definizione della variabile accumulatore */
   int i=0; /* variabile contatore */

   while (i!=n) {
      i++;
      f *= i;
   }
   return f;
}
```

Esiste anche un'altra definizione, quella *induttiva*, che si limita a definire il valore della funzione fattoriale su 0 e il valore del fattoriale di un intero $n+1$ in funzione del fattoriale di n:

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) \times n! \end{aligned}$$

Una proprietà chiave dei numeri naturali è quella che definizioni di questo tipo effettivamente individuano in modo univoco una funzione.

La versione ricorsiva è sostanzialmente una traduzione in `Java` della definizione induttiva del fattoriale. Vedremo altri esempi in cui un programma `Java` sarà analogo ad una definizione induttiva.

```
public static int fattRec(int n) {
   if (n==0) return 1;     /* caso base */
   return n*fattRec(n-1); /* passo induttivo */
}
```

Le equazioni ricorsive che definiscono il fattoriale sono semplicemente state tradotte in `Java`: i diversi casi della definizione vengono discriminati da un costrutto di tipo `if` e abbiamo sostituito la notazione postfissa del simbolo di fattoriale ('!') con le chiamate ricorsive al metodo `fattRec`.

Da un punto di vista concreto i due metodi appena mostrati eseguono esattamente le stesse operazioni (n prodotti). La versione iterativa sarà leggermente più efficiente, mentre la versione ricorsiva è più vicina alla definizione matematica e mostra meno dettagli implementativi (variabili contatori e accumulatori).

Nel caso specifico non è particolarmente difficile capire cosa calcola il metodo `fattIt`, ma esistono esempi in cui la versione iterativa di una corrispondente funzione ricorsiva non è affatto semplice da trovare (ad esempio il ben noto problema della Torre di Hanoi). Questo perchè i programmi ricorsivi mimano una naturale forma di ragionamento matematico, l'induzione, che é un modo naturale di risolvere un problema: studiare i casi semplici (casi base) e ridurre la soluzione di istanze complicate a quella di istanze più semplici (passo induttivo). A volte, tuttavia, la maggior semplicità del programma ricorsivo nasconde una complessità gestita implicitamente dal meccanismo computazionale che implementa la ricorsione, come vedremo nella prossima sezione.

1.2 La funzione fibonacci

La *successione di fibonacci*[1] viene induttivamente definita come segue:

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \quad \text{per } n > 1 \end{aligned}$$

Essa definisce la successione di interi, $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. Di questa successione non è altrettanto evidente dare una definizione informale.

[1] La successione prende il nome dal matematico italiano Leonardo Pisano detto FIBONACCI (Pisa ∼1170-∼1250) che l'ha introdotta per studiare la riproduzione dei conigli, dando risposta alla seguente domanda: partendo da una coppia di conigli e supponendo che ogni coppia di conigli produca una nuova coppia ogni mese, a partire dal secondo mese di vita, quante coppie di conigli ci sono dopo n mesi? La successione di Fibonacci gode di numerose proprietà interessanti ed ha trovato successivamente moltre altre applicazioni in matematica.

Proponiamoci ora di scrivere due metodi, uno ricorsivo e uno iterativo, che, preso in ingresso un intero n, calcolino l'n-esimo numero della successione di fibonacci. Il metodo ricorsivo si può scrivere facilmente semplicemente traducendo le equazioni ricorsive in `Java`, come nel caso del fattoriale:

```
public static int fibRec(int n) {
   if (n<=1) return n;               /* caso base */
   return fibRec(n-1) + fibRec(n-1); /* passo induttivo */
}
```

Il metodo iterativo è leggermente più complicato: tuttavia per scriverlo è sufficiente osservare che per calcolare l'n-esimo numero di fibonacci è necessario conoscere i due precedenti numeri di fibonacci. Visto che conosciamo i primi due numeri della serie, possiamo pensare di calcolarli tutti a partire dal terzo fino a quello desiderato. L'unica avvertenza è quella di mantenere sempre memorizzati gli ultimi due numeri calcolati per trovare il successivo numero di fibonacci.

```
public static int fibIt(int n) {
  int fib1=0; /* variabile per fibonacci di n-1 */
  int fib2=1; /* variabile per fibonacci di n-2 */
  int fibnew; /* variabile per memorizzare il nuovo numero calcolato */
  int i = 2;

  if (n<2) return n;
  while (i<=n) {
    fibnew = fib1 + fib2;
    fib2 = fib1;
    fib1 = fibnew;
    i++;
  }
  return fib1;
}
```

A differenza del caso del fattoriale, le due funzioni `fibRec` e `fibIt`, si comportano in modo molto diverso: il metodo`fibIt` esegue il ciclo esattamente $n-2$ volte e ciascun ciclo costa una somma e 3 assegnazioni. Il metodo ricorsivo per calcolare l'n-esimo numero di fibonacci, viceversa, invoca il calcolo dell'$(n-1)$-esimo e dell'$(n-2)$-esimo. A sua volta il calcolo dell'$(n-1)$-esimo numero invocherà il calcolo dell' $(n-2)$-esimo e dell'$(n-3)$-esimo: già a questo punto è chiaro che parte del lavoro viene ripetuto inutilmente. In Fig. 1.2 viene esemplificato l'albero delle chiamate ricorsive generato per effetto di una chiamata a `fibRec(4)`.

E' facile dimostrare (per induzione!) che il calcolo di $fib(1)$ e $fib(0)$ verranno invocati rispettivamente $fib(n)$ e $fib(n-1)$ volte (vedi figura 1.2), che è un numero che cresce esponenzialmente; quindi in questo caso la semplicità del metodo ricorsivo viene pagata a caro prezzo in termini di efficienza. Per convincersi di ciò in maniera empirica, possiamo definire una classe `Fibonacci` che fornisce come metodi `fibRec` e `fibIt`; nel main lanciamo i due metodi e ne valutiamo le prestazioni stampando il tempo macchina necessario per calcolare il fattoriale dello stesso numero con i due metodi:

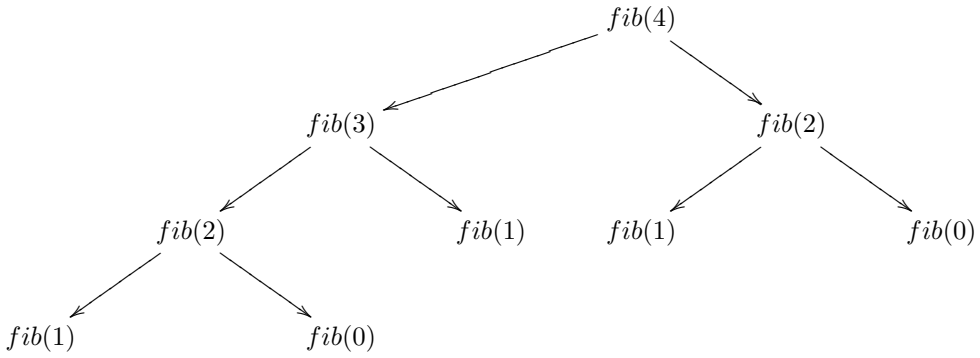

Figura 1: Attivazione delle chiamate ricorsive del metodo `fibRec(4)`

```
public class Fibonacci {
   public static int fibRec(int n) { ... }

   public static int fibIt(int n) { ... }

   public static void main (String[] args) {
      long time = System.currentTimeMillis();
      fibRec(30);
      time = System.currentTimeMillis() - time;
      System.out.println("Versione ricorsiva: " + time + " millisecondi");
      time = System.currentTimeMillis();
      fibIt(30);
      time = System.currentTimeMillis() - time;
      System.out.println("Versione ricorsiva: " + time + " millisecondi");
   }
}
```

Sulla mia macchina, la versione iterativa ha impiegato meno di 1 millisecondo, quella ricorsiva oltre 5 secondi!!

Ma non si può proprio fare meglio di così? Nel senso, è proprio necessario dover scegliere tra efficienza di esecuzione e intuitività di programmazione? Mostreremo ora come un po' di furbizia programmativa può combinare insieme l'efficienza di un numero lineare di chiamate ricorsive con l'intuitività della programmazione basata su ricorsione. L'idea di base è quella di mimare ricorsivamente il comportamento del programma iterativo. Dovremo quindi usare la ricorsione per mimare il comportamento del ciclo:

```
while (i<=n) {
   fibnew = fib1 + fib2;
   fib1 = fib2;
   fib2 = fibnew;
   i++;
}
```

Per quanto riguarda il controllo, è facile rendersi conto che è sufficiente introdurre tra i parametri del metodo ricorsivo un parametro che gioca il ruolo dell'indice `i` e mantenere un parametro col valore di ingresso `n`, e:

1. terminare le attivazioni ricorsive quando il valore di `i` raggiunge il valore di `n`;
2. incrementare ad ogni attivazione ricorsiva il valore del parametro `i`

L'altro problema riguarda come mantenere lo stato della computazione, cioè come sia possibile mantenere i valori via via computati e memorizzati nelle variabili `fib1` e `fib2` e `fibnew`. Ricordiamo che le variabili dichiarate in un metodo sono *locali* e quindi per ciascuna variabile viene riallocata una nuova copia ad ogni attivazione ricorsiva del metodo e di conseguenza eventuali assegnazioni fatte durante un'attivazione **non** influenzano il valore di quelle variabili in una successiva attivazione, o al rientro da quella attivazione. Una pessima soluzione sarebbe quella di dichiarare `fib1 e fib2` e `fibnew` come variabili statiche: questo condurrebbe ad un fenomeno detto *side effect* assolutamente sconsigliato, in quanto un metodo che modifica lo stato globale in modo non specificato dall'interfaccia rende l'analisi dei programmi molto difficile. Ma allora come comunicare i valori via via calcolati alle nuove attivazioni? La risposta è semplice: usare i parametri. Dovremmo quindi arricchire l'interfaccia del metodo con dei parametri ausiliari che giocheranno un ruolo analogo alle variabili `fib1` e `fib2` nel metodo iterativo.

Siccome però vorremmo scrivere un metodo `fibRecEff` che accetta in input *un solo* parametro intero (e che quindi abbia la stessa interfaccia dei metodi `fibRec` e `fibIt`), scriveremo un metodo ausiliario privato `fibRecEffAux` con i parametri necessari a mimare le operazioni del metodo iterativo. Il metodo `fibRecEff` si limiterà a trattare i casi base e chiamare il metodo ausiliario `fibRecEffAux` inizializzando opportunamente i parametri per la prima chiamata, esattamente come il metodo iterativo `fibIt` inizializza opportunamente le variabili `i, fib1` e `fib2` all'ingresso del ciclo.

Ecco quindi il codice dei due metodi:

```
private static int fibRecEffAux(int n, int i, int fib1, int fib2) {
   if (i==n) return fib1;
   return fibRecEffAux(n,i+1,fib1+fib2, fib1);
}

public static int fibRecEff(int n) {
   if (n<2) return n;
   return fibRecEffAux(n,2,1,0);
}
```

Invocando ora il metodo `fibRecEff(30)` dentro il main e confrontando i tempi di sistema, ci si accorge che la verisone iterativa e quella ricorsiva efficiente hanno prestazioni paragonabili. Va notato che eseguendo più volte il test, magari su input diversi, non necessariamente la versione iterativa sarà più efficiente (anche se di poco) di quella ricorsiva. Questo è dovuto sia al fatto che non sappiamo bene cos'altro sta facendo la macchina mentre svolgiamo i vari test, sia (soprattutto) perché il compilatore effettua procedure di ottimizzazione del codice per dar vita a programmi ricorsivi efficienti, se possibile (come lo è nel caso di `fibRecEffAux`).

1.3 Ricerca Binaria

Il ben noto problema proposto in questa sezione consiste nel trovare un elemento all'interno di un *vettore ordinato* restituendo la sua posizione se viene trovato, oppure -1 in caso contrario. L'algoritmo di *ricerca binaria* sfrutta la seguente idea:

- controllo l'elemento mediano
- se questo è l'elemento cercato, allora ho finito
- se è minore dell'elemento cercato, allora devo andare in cerca nella parte destra
- altrimenti nella parte sinistra.

In qualche modo, questo algoritmo è analogo al modo con cui cerchiamo una parola nel dizionario[2].

L'algoritmo di ricerca binaria si esprime in modo estremamente naturale in modo ricorsivo: supponiamo di sapere che l'elemento da cercare stia tra *inf* e *sup*. Trova il punto medio m. Se $A[m]$ e' l'elemento giusto, ritorna m altrimenti cerca nuovamente nell'intervallo $[m+1, sup]$ se $A[m] < elem$ oppure nell'intervallo $[inf, m-1]$ se $A[m] > elem$. Vediamo il codice:

```
public static int ricBinRec(int[] A, int inf, int sup, int elem) {
  int m;

  if (inf>sup) return -1;
  m = (inf + sup) / 2;
  if (elem==A[m]) return m;
  if (elem<A[m]) return ricBinRec(A, m+1, sup, elem);
  else return ricBinRec(A, inf, m-1, elem);
}
```

In questo caso, la funzione va invocata scrivendo `ricBinRec(V, 0, V.length, e)`, assumendo che `V` sia il vettore ed `e` l'elemento da cercare.

La traduzione in uno stile iterativo è anche in questo caso abbastanza facile, e l'efficienza delle due versioni è paragonabile:

```
public static int ricBinIt(int[] A, int elem) {
  int m, inf = 0, sup = A.length-1;

  if (elem<A[inf] || elem>A[sup]) return -1;
  while (inf <= sup) {
    m = (inf+sup) / 2;
    if (elem==A[m]) return m;
    if (elem<A[m]) inf=m+1; else sup=m-1;
  }
  return -1;
}
```

[2]in raltà un essere umano fa delle ipotesi iniziali più forti su dove probabilmente la parola si trova, basate sull'esperienza di precedenti consultazioni e quindi si dirige sicuro verso la fine se ad esempio cerca una parola che comincia con la lettera "t"

Notiamo che il comando:

```
if (elem<A[inf] || elem>A[sup]) return -1;
```

è di fatto inutile, e la funzione darebbe comunque risultati corretti. Tra breve avremo i mezzi formali per dimostrare ciò.

2 Introduzione all'uso di asserzioni logiche

A questo punto risulta interessante chiedersi: è possibile dimostrare che le funzioni scritte calcolano esattamente quanto desiderato? Cosa succede quando vengono forniti input incoerenti (ad esempio viene chiesto di calcolare il fattoriale di un numero negativo)?

Vedremo che sarà opportuno completare la specifica di una procedura con dei commenti che esprimono cosa viene calcolato da una procedura (*postcondizioni*), sotto opportune assunzioni sui dati in ingresso (*precondizioni*): questo automaticamente, vedremo, fornisce una tecnica di prova per dimostrare la correttezza delle funzioni ricorsive. Infine vedremo una tecnica di prova per i programmi iterativi (*invarianti*).

2.1 Precondizioni

Ritorniamo ai codici dei metodi `fattIt` e `fattRec`. Cosa accade qualora il parametro di ingresso sia negativo? In entrambi i casi, i metodi non terminano, in quanto in `fattIt` il parametro `i` viene inizializzato a 0 e continua a crescere non raggiungendo mai il valore negativo `n`, mentre `fattRec` continua a chiamare se stessa su valori più piccoli senza incontrare mai la condizione di chiusura della chiamate ricorsive, causando probabilmente un errore di sistema (*stack overflow*).

Si tratta di un comportamento anomalo ed indesiderato, ma prevedibile visto che il fattoriale è definito solo su interi non-negativi. I due metodi funzionano correttamente solo quando è verificata la asserzione logica $n \geq 0$. Un'asserzione che specifica le proprietà che si devono verificare al momento dell'attivazione di un metodo affinché questo dia i risultati desiderati si chiama *precondizione* oppure *asserzione iniziale*. Scriveremo le precondizioni come commenti dentro il codice sorgente dei programmi, come segue:

```
public static int fattRec(int n) {
/* PREC: n >= 0 */
   if (n==0) return 1;        /* caso base */
   return n*fattRec(n-1);     /* passo induttivo */
}
```

In questo modo il programmatore che definisce `fattRec` specifica, oltre che al suo codice, anche il suo corretto utilizzo[3]: in tal modo un programmatore utente della definizione di `fattRec` è informato che dovrà evitare chiamate scorrette che non rispettano la precondizione. Non solo, ma anche il programmatore di `fattRec` dovrà preoccuparsi che le chiamate ricorsive rispettino la precondizione: nel nostro caso, nell'ipotesi $n \geq 0$ la chiamata ricorsiva verrà effettuata con un parametro che soddisfa la stessa proprietà: infatti se $n = 0$ non attivo alcuna chiamata ricorsiva, mentre $n > 0$ implica $n - 1 \geq 0$.

[3]In realtà sarebbe possibile evitare la non terminazione, usando la condizione `n<=0`: tuttavia in tal caso, benchè il metodo termini sempre, i risultati ottenuti sui numeri negativi (verrebbe restituito sempre 1 se il parametro di ingresso è negativo) sono comunque arbitrari, visto che la funzione fattoriale è definita solo sugli interi positivi

2.2 Postcondizioni

L'altra clausola contrattuale a cui deve adempiere chi definisce un metodo è la specifica di quanto il metodo calcola, ovviamente sotto le ipotesi descritte nella precondizione. Questa asserzione logica viene detta *postcondizione* oppure *asserzione finale*. Scriveremo anche queste dentro il testo del programma sotto forma di commento:

```
public static int fattRec(int n) {
/* PREC: n >= 0
 * POST: ritorna n!
 */
    if (n==0) return 1;       /* caso base */
    return n*fattRec(n-1);    /* passo induttivo */
}
```

Osserviamo ora come sia semplice valutare la correttezza di una funzione ricorsiva: basterà fare una semplice dimostrazione per induzione. Nel nostro caso dobbiamo dimostrare che, ricevendo in input un intero positivo, `fattRec` ne restituisce il fattoriale. Cioè che, sotto le ipotesi della precondizione, il metodo garantisce la postocondizione, assumendo che le chiamate ricorsive restituiscano valori corretti (bisognerà comunque verificare che le chiamate ricorsive rispettano la precondizione). La base di induzione consiste nella banale verifica che per $n = 0$ viene restituito 1, come stabilito dalla definizione di fattoriale. Il passo induttivo consiste nel verificare che per $n > 0$ viene restituito $n!$. Ma ciò è banale perché per $n > 0$ viene attivata la chiamata ricorsiva a `fattRec(n-1)`: questa attivazione soddisfa le precondizioni, in quanto $n > 0$ implica $n - 1 \geq 0$; possiamo quindi usare l'ipotesi induttiva, cioè che la chiamata restituisce $(n - 1)!$. A questo punto, è sufficiente osservare che `fattRec` restituisce $n \times (n - 1)!$, ma per definizione di fattoriale questo è esattamente $n!$.

2.3 Invarianti

Vediamo infine come sia possibile stabilire la correttezza di un programma iterativo. Un ciclo produce la ripetizione di un blocco di comandi, fino al verificarsi di una condizione: di conseguenza si tratta di ripetere la stessa computazione (al variare eventualmente del valore di alcune variabili): quindi è naturale pensare che durante l'esecuzione di un ciclo ci siano delle proprietà che rimangono invarianti (e che anch'esse verosimilmente dipendono dal valore delle variabili che vengono modificate durante il ciclo).

Un'*invariante* è una asserzione logica che esprime una proprietà sempre verificata durante l'esecuzione di un ciclo. Vediamo come si può valutare la correttezza di un ciclo, riprendendo le versioni iterative delle funzioni che calcolano rispettivamente il fattoriale e l'ennesimo numero della successione di fibonacci. Scriveremo anche le invarianti, così come le altre annotazioni logiche (precondizioni e postcondizioni), come commenti all'interno dei programmi. Ecco di nuovo la funzione fattoriale opportunamente annotata:

```
public static int fattIt(int n) {
/* PREC: n >= 0
 * POST: ritorna n!
 */
   int f=1, i=0;
```

```
   while (i<n) {
   /* INV: f=i! */
      i++;
      f *= i;
   }
return f;
}
```

Informalmente è facile osservare che prima dell'esecuzione del primo ciclo, la variabile `f` contiene il valore 1 ed `i` il valore 0. Quindi, la proprietà invariante è verificata all'ingresso del ciclo. Ad ogni iterazione, se all'inizio del ciclo la variable `f` contiene il valore `i!`, si incrementerà di 1 la variabile `i` ed `f` viene riaggiornata opportunamente.

Concludiamo questa sottosezione vedendo il programma iterativo che calcola l'ennesimo numero di fibonacci, opportunamente annotato.

```
public static int fibIt(int n) {
/* PREC: n >= 0
 * POST: restituisce l'n-esimo numero di fibonacci
   int fib1=1, fib2=0, finbew=1, i = 2;

   if (n<2) return n;
   while (i<=n) {
   /* INV: fib1 = fibnew = fib(i-1) & fib2 = fib(i-2) */
      fibnew = fib1 + fib2;
      fib2 = fib1;
      fib1 = fibnew;
      i++;
   }
   return fib1;
}
```

È ora facile dimostrare formalmente la correttezza del programma.

2.4 Terminazione

Una proprietà auspicabile dei programmi è la *terminazione*[4]. Un programma termina se dopo un numero finito di passi la sua computazione termina. L'esempio più banale di programma non terminante è il seguente:

```
public static void loop () {
   while (true) {};
}
```

[4]In realtà, esistono alcune applicazioni in cui è utile considerare processi che non terminano, ad esempio nel campo dei sistemi operativi o dei controlli dei sistemi. Pensate banalmente ad un qualsiasi programma che gestisce l'interazione col calcolatore. Anche in questi casi, tuttavia, ci sono nozioni simili alla terminazione: ad esempio, è auspicabile che un programma, anche non terminante, comunque dopo un numero finito di passi torni in uno stato in cui interagisce con i dispositivi di input/output.

che è il programma che non fa nulla per l'eternità. In generale, un programma può fare un numero *illimitato* di passi, e questo non prevedibile a priori[5]. Vedremo ora una semplice tecnica per dimostrare che un programma termina, applicabile sia nel caso dei cicli che nel caso delle funzioni ricorsive. Occupiamoci del primo caso, lasciando il secondo caso alle riflessioni del lettore.

Per dimostrare che un ciclo termina, è sufficiente trovare una funzione, detta *funzione di terminazione*, dipendente dallo stato della computazione (dipenderà tipicamente dal valore delle variabili modificate nel corpo del ciclo e/o coinvolte nella guardia) sempre positiva e sempre decrescente durante il ciclo. Più formalmente:

Definizione 2.1 [Funzione di Terminazione] Sia φ una proprietà invariante per il ciclo `while (b) C`; t è detta *funzione di terminazione* se gode delle seguenti proprietà (indichiamo con v il valore risultante dalla valutazione della guardia del ciclo):

1. $\varphi \wedge v \Rightarrow t \geq 0$;
2. t decresce ad ogni iterazione.

Teorema 2.2 *Se esiste una funzione di terminazione t per il ciclo* `while (b) C` *allora il ciclo termina dopo un numero finito di iterazioni.*

Dim: Supponiamo per assurdo che esista una funzione di terminazione t ma che il ciclo non termini. Allora, indicando con S_i $(i \in \mathbb{N})$ lo stato all'i-esima iterazione, si avrebbe una successione infinita strettamente decrescente di numeri naturali $t(S_0) > t(S_1) > \ldots t(S_n) > t(S_{n+1}) > \ldots > 0$. Il che è assurdo, perché nei numeri naturali non esistono sequenze infinite strettamente decrescenti. ✓

Proviamo a definire una funzione di terminazione per `fattIt`. Prendiamo $t(n, i) = n - i$; tale funzione è una funzione di terminazione per il ciclo in esame. Infatti, se sono vere l'invariante di ciclo (`INV: f = i!`) e la guardia del ciclo (`i < n`), abbiamo che $n - i \geq 0$ (in realtà, ciò è assicurato dalla sola guardia; l'invariante non gioca qui alcun ruolo); inoltre, passando dall'iterazione i-esima alla $(i+1)$-esima, il valore di t passa da $n - i$ a $n - (i+1) = n - i - 1$, cioè decresce strettamente. Questo basta a garantire la terminazione di `fattIt` su ogni input che rispetta la precondizione. Il fatto che l'input rispetta la precondizione è garantito dalla validità dell'invariante: siccome deve essere che $f = i!$, abbiamo che $i \geq 0$, visto che il fattoriale è definito solo sui naturali; visto che $i < n$ (e questo vale perché è vera la guardia del ciclo), deve essere $n > 0$.

La terminazione per metodi definiti ricorsivamente si studia in maniera analoga. In tal caso, però, la funzione di terminazione dipende, invece che dallo stato della memoria, dai parametri passati al metodo, e in particolare da quelli che occorrono nelle condizioni che regolano l'attivazione di nuove chiamate ricorsive. Inoltre, non essendoci invarianti di ciclo, la funzione di terminazione sarà definita in base alla precondizione. Per esempio, nel caso di `fattRec`, basta prendere $t(n) = n$: se è vera la precondizione, allora $t(n) = n \geq 0$; inoltre, ad ogni chiamata il valore dell'argomento decresce di 1.

[5] In generale infatti, non è possibile scrivere un programma che preso in input un altro programma, dica se questo termina o meno.

3 Esempi

3.1 Versione efficiente per il calcolo ricorsivo dei numeri di fibonacci

Iniziamo con l'annotare il codice con le opportune asserzioni logiche:

```
public static int fibRecEffAux(int n, int i, int fib1, int fib2) {
/* PREC: fib1=fib(i) & fib2=fib(i-1) & 2<=i<=n
 * POST: ritorna fib(n) */
   if (i==n) return fib1;
   return fibRecEffAux(n,i+1,fib1+fib2, fib1);
}

public static int fibRecEff(int n) {
/* PREC: n>=0
 * POST: ritorna fib(n) */
   if (n<2) return n;
   return fibRecEffAux(n,2,1,0);
}
```

Dimostriamo ora per induzione che una chiamata alla funzione `fibRecEff(n)` effettivamente restituisce $fib(n)$, sotto la precondizione $n \geq 0$. Per $n < 2$ è sufficiente verificare che $n = fib(n)$ ed effettivamente la funzione restituisce n. Viceversa la correttezza di `fibRecEff` dipende dalla correttezza di `fibRecEffAux`. Dimostriamo le seguenti cose:

1. La chiamata iniziale `fibRecEffAux(n,2,1,0)` rispetta le precondizioni. Infatti, essendo falsa la condizione dell'`if` del metodo `fibRicEff`, $n \geq 2$ e quindi $2 = i \leq n$. Inoltre $fib(2) = 1$ e $fib(1) = 0$;

2. se $i = n$ e vale la precondizione ($fib(i) = fib1$), chiaramente la funzione restituisce $fib(n)$;

3. altrimenti, induttivamente la correttezza di `fibRecAuxEff(n,i,fib1,fib2)` dipende solo dalla correttezza di `fibRecAuxEff(n,i+1,fib1+fib2, fib1)`. L'unica cosa da far vedere è che la nuova chiamata rispetta le precondizioni, ma:

   - $i \leq n \wedge i \neq n$ implica $i + 1 \leq n$;
   - per definizione della funzione di fibonacci, se $fib1 = fib(i) \wedge fib2 = fib(i-1)$, allora $fib1 + fib2 = fib(i+1)$.

4. infine, osserviamo che la sequenza di chiamate ricorsive termina, perché la funzione $n - i$ (positiva sotto le precondizioni) decresce ad ogni chiamata.

Concludiamo con un ulteriore spunto di riflessione. La funzione `fibRecEff`, pur facendo circa le stesse operazioni di `fibIt`, sarà comunque più inefficiente. Questo perché il passaggio di parametri e l'allocazione dei record di attivazione hanno un costo.

3.2 Ricerca Binaria

Mettiamoci nell'ipotesi che l'elemento cercato abbia un valore compreso tra il minimo ed il massimo del vettore (ciò è verificabile in tempo costante prima della chiamata del metodo); siccome il vettore è per ipotesi ordinato, avrò che è verificata la seguente relazione (A è il nome del vettore, n la sua lunghezza ed $elem$ l'elemento cercato):

$$A[0] \leq elem \leq A[n-1] \tag{1}$$

Ricordiamo che le due variabili inf e sup delimitano l'intervallo di ricerca ad $A[inf], ..., A[sup]$. Vogliamo continuare la ricerca assicurandoci che valga l'asserzione:

$$A[inf] \leq elem \leq A[sup] \tag{2}$$

che chiaramente è un'ottima candidata ad essere l'invariante del nostro ciclo.

L'ipotesi semplificativa (1) è facilmente assicurabile, anche se non prevista dalle precondizioni. Infatti se $elem < V[0]$ oppure $elem > V[n-1]$ possiamo concludere subito, visto che V è ordinato. Avendo la proprietà (1), possiamo inizializzare una variabile `inf` a `0` e una `sup` a `n-1`, che soddisfa (2).

Rimane il problema di come riaggiornare gli estremi dell'intervallo di ricerca; innanzitutto cerchiamo il punto medio. A patto che $inf \leq sup$, il comando: `m = (inf + sup) / 2` calcola il punto medio usando la divisione intera (se la variabile `m` è dichiarata intera). Osserviamo che la condizione $inf \leq sup$ esprime il fatto che il nostro intervallo di ricerca non sia vuoto. Si tratta quindi di una perfetta candidata al ruolo di guardia del ciclo. Se $A[m] = elem$, ho chiaramente finito la ricerca e restituisco il valore della variabile `m`. Altrimenti è sufficiente osservare che, sotto le ipotesi che il vettore sia ordinato:

$$\begin{array}{rcl} (A[m] < elem) \wedge (A[inf] \leq elem \leq A[sup]) & \Rightarrow & A[m] < elem \leq A[sup] \\ (A[m] > elem) \wedge (A[inf] \leq elem \leq A[sup]) & \Rightarrow & A[inf] \leq elem < A[m] \end{array}$$

E quindi, usando nel primo caso l'assegnazione `inf = m+1` e nel secondo `sup = m-1` mantengo la proprietà (2).

Rimangono due problemi. La terminazione e come capire se siamo usciti dal ciclo perché abbiamo trovato l'elemento o perché abbiamo verificato che l'elemento non c'è. La terminazione si dimostra semplicemente prendendo la funzione $t = sup - inf$. E' ovvio che: $inf \leq sup \Rightarrow t = \sup - inf \geq 0$. Inoltre, le seguenti due relazioni valgono:

$$sup - (m+1) < sup - inf \qquad (m-1) - inf < sup - inf$$

Infatti,

$$sup - \left(\frac{inf + sup}{2} + 1\right) = \frac{sup - inf - 2}{2} < \frac{sup - inf}{2} \leq sup - inf$$

e

$$\left(\frac{inf + sup}{2} - 1\right) - inf = \frac{sup - inf - 2}{2} < sup - inf$$

Ciò implica che la funzione t è strettamente decrescente sotto le condizioni sempre verificate all'interno del ciclo.

Chiaramente la condizione $inf > sup$ implica che siamo usciti dal ciclo perchè non abbiamo trovato l'elemento ed abbiamo esaurito lo spazio di ricerca. Siamo finalmente pronti a scrivere il programma completo:

```
public static int ricBinIt(int A[], int elem) {
/* PREC: ord(A)
 * POST: ritorna k se A[k]=elem, -1 altrimenti
 */
   int m, inf = 0, sup = A.length-1;

   if (elem<A[inf] || elem>A[sup]) return -1;
   while (inf <= sup) {
   /* INV: A[inf]<=elem<=A[sup]
    * TERM: t=sup-inf */
      m = (inf+sup) / 2;
      if (elem==A[m]) break;
      else if (elem<A[m]) inf=m+1;
           else sup=m-1;
   }
   if (inf>sup) return -1 else return m;
}
```

Il lettore è invitato a dimostrare che il comando:

```
if (elem<A[inf] || elem>A[sup]) return -1;
```

è di fatto inutile, e la funzione darebbe comunque risultati corretti. La dimostrazione segue gli stessi passi, usando però il seguente invariante un po' più complicato:

$$\exists k(A[k] = elem \Rightarrow inf \leq k \leq sup)$$

che esprime il fatto che se *elem* c'è nel vettore, allora l'indice che occupa sta nell'intervallo che abbiamo selezionato.

Come abbiamo già visto, l'algoritmo di ricerca binaria si esprime in modo estremamente naturale in modo ricorsivo. Anche in questo caso, correttezza e terminazione si dimostrano usando asserzioni:

```
public static int ricBinRec(int A[], int inf, int sup, int elem) {
/* PREC: ord(A), se esiste k t.c. A[k]=elem allora inf<=k<=sup
 * POST: ritorna k se A[k]=elem, -1 altrimenti
 */
   int m;

   if (inf>sup) return -1;
   m = (inf + sup) / 2;
   if (elem==A[m]) return m;
   if (elem<A[m]) return ricBinRec(A, m+1, sup, elem);
   else return ricBinRec(A, inf, m-1, elem);
}
```

I ragionamenti necessari per dimostrare la correttezza del metodo ricorsivo non sono molti diversi da quelli fatti per il metodo iterativo, solo che saranno leggermente più semplici.

3.3 Selection Sort

Vediamo in questa sezione una ben nota procedura di ordinamento di vettori, detta *Selection Sort*, basata sul principio di trovare i minimi successivi della parte di vettore non ancora ordinato e piazzarli via via in testa al vettore già ordinato. Al generico passo i, la situazione è la seguente:

1. il vettore è ordinato fino alla posizione $i-1$. Formalmente:

$$\varphi[i] \equiv \forall j(0 \leq j < i \Rightarrow A[j] \leq A[j+1])$$

2. tutti gli elementi del vettore dalla posizione i in poi sono maggiori degli elementi che stanno fino alla posizione $i-1$. Formalmente:

$$\psi[i] \equiv \forall k, j((0 \leq j < i \wedge i+1 \leq k < n) \Rightarrow A[j] \leq A[k])$$

A questo punto, conoscendo l'indice k_0, tale che $A[k_0] = min_{i \leq k < n} A[k]$, è chiaro che, scambiando $A[k_0]$ con $A[i]$, avremo soddisfatte le proprietà $\varphi[i+1]$ e $\psi[i+1]$, perché $A[k_0]$ è maggiore di tutti gli elementi fino a $i-1$ (per $\psi[i]$) e, per come è stato scelto, è minore di tutti gli elementi da $i+1$ in poi.

Per implementare questo in maniera semplice, usiamo un metodo (privato) ausiliario che restituisce l'indice del minimo elemento in un array non ordinato:

```
private static int minimo(int[] A, int inf, int sup) {
/* PREC: sup > inf >= 0
 * POST: ritorna k t.c., per ogni inf <= j <= sup, A[k] <= A[j]
 */
   int i = inf, k = inf;

   while (i <= sup) {
   /* INV: forall j, se inf <= j < i allora A[k] <= A[j] */
      if (A[i] < A[k]) k=i;
      i++;
   }
   return k;
}
```

E' relativamente facile verificare che l'invariante è verificato all'entrata del ciclo. Infatti, non esistono j tali che $inf \leq j < i = inf$ e un'implicazione con la guardia falsa è sempre vera. Altrettanto evidente è che l'invariante e la negazione della guardia implicano l'asserzione finale. Una funzione di terminazione può essere $sup - i$: come nel caso del fattoriale, si mostra che decresce ad ogni iterazione e che assume valori non-negativi se la guardia del ciclo è verificata. Resta da verificare che l'invariante viene mantenuto dalla sequenza di comandi contenuta nel corpo del ciclo. Se la guardia dell'`if` è soddisfatta, allora k assume il valore i; da ciò e dalla validità dell'invariante, banalmente segue che

$$\forall j((inf \leq j \leq i) \Rightarrow A[k] \leq A[j])$$

Se invece la guardia dell'`if` non è soddisfatta, allora k mantiene il valore che aveva; da ciò e dalla validità dell'invariante, nuovamente segue che

$$\forall j((inf \leq j \leq i) \Rightarrow A[k] \leq A[j])$$

Pertanto, a seguito dell'incremento di i (cioè, al termine del corpo del `while`), torniamo in una situazione in cui l'invariante è verificata.

Possiamo ora scrivere la procedura di ordinamento `selectSort` come segue:

```
public static void selectSort (int[] A) {
/* PREC:
 * POST: forall j, se 0 <= j < A.length-1 allora A[j] <= A[j+1]
 */
   int i = 0;
   while (i < A.length-1) {
   /* INV: forall j,k se 0 <= j < i e i <= k < n
    *                 allora A[j] <= A[j+1] e A[j] <= A[k]*/
     scambia(A, i, minimo(A,i,A.length-1));
     i++;
   }
}
```

Assumiamo anche un metodo `private void scambia(int[] A, int i, int j)` che scambia gli elementi di indice i e j nel vettore A. Tale metodo è molto semplice e può essere dimostrato corretto in maniera banale. Non banale, ma comunque facile, è la verifica formale che $\varphi[i] \wedge \psi[i]$ è una proprietà invariante, mentre è chiaro che $\varphi[i] \wedge (i = n-1)$ implica la postcondizione. Sempre il solito ragionamento (e la solita funzione di terminazione $n - i$) permettono di dedurre che il ciclo termina. Lasciamo al lettore questi compiti per esercizio.

Il lettore attento avrà osservato che il metodo `selectSort`, pur avendo due cicli annidati (nel codice scritto questo è nascosto dalla chiamata a `minimo`), non è più difficile da analizzare rispetto altri programmi nelle sezioni precedenti. Questo perchè abbiamo convenientemente scomposto il problema in sottoproblemi e scritto ed analizzato il metodo assumendo che un altro metodo fosse corretto: non solo i ragionamenti formali sono facilitati dalla scomposizione di un problema complesso in sotto-problemi, ma anche quelli informali: in generale una buona tecnica di programmazione consiste nel suddividere un problema in sottoproblemi, risolverli separatamente e poi comporli in modo adeguato per risolvere il problema di partenza (metodologia *top-down*).

Esiste anche un atteggiamento diametralmente opposto, ma animato dallo stesso principio (noto come *divide et impera*): costruire nuovi tipi di dato e operazioni sempre più complesse per fornire ad altri programmatori delle azioni elementari via via più complesse e quindi adatte a risolvere problemi complessi in modo relativamente semplice (metodologia *bottom-up*). Sarebbe particolarmente importante che il comportamento degli strumenti software forniti ad altri programmatori sia specificato in modo preciso e non ambiguo, attraverso asserzioni logiche.